

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

APPLICATION FOR LETTERS PATENT

**Methods, Systems, Architectures and Data Structures  
For Delivering Software via a Network**

Inventor(s):

Sarita James

Brian Syme

Suryanarayanan Raman

Lawrence Sanchez

John Licata

## RELATED APPLICATIONS

The following patent applications are related to the present application, are assigned to the assignee of this patent application, and are expressly incorporated by reference herein:

- U.S. Patent Application Serial No. \_\_\_\_\_, entitled "Single Window Navigation Methods and Systems", bearing attorney docket number MS1-560us, and filed on the same date as this patent application;
- U.S. Patent Application Serial No. \_\_\_\_\_, entitled "Methods and Systems of Providing Information to Computer Users", bearing attorney docket number MS1-557us, and filed on the same date as this patent application;
- U.S. Patent Application Serial No. \_\_\_\_\_, entitled "Network-based Software Extensions", bearing attorney docket number MS1-563us, and filed on the same date as this patent application;
- U.S. Patent Application Serial No. \_\_\_\_\_, entitled "Authoring Arbitrary XML Documents Using DHTML and XSLT", bearing attorney docket number MS1-583us, and filed on the same date as this patent application;
- U.S. Patent Application Serial No. \_\_\_\_\_, entitled "Architectures For And Methods Of Providing Network-based Software Extensions", bearing attorney docket number MS1-586us, and filed on the same date as this patent application.
- U.S. Patent Application Serial No. \_\_\_\_\_, entitled "Task Sensitive Methods And Systems For Displaying Command Sets", bearing attorney docket number MS1-562us, and filed on the same date as this patent application.

## TECHNICAL FIELD

This invention relates to methods and systems for providing software via a network. More specifically, the invention pertains to Internet-based delivery of software.

## 1

2

2

1

client. The program or platform is configured so that it is extensible based on software extensions that are deliverable over a network such as the Internet. Various extensions can be developed by third party developers for incorporation into the program or platform.

In one described embodiment, extension files that comprise a software extension are hosted on a network server such as an Internet server. These include descriptor files that describe aspects of the software extension. These descriptor files logically describe an extension to the program or platform and designate the location of other extension files. Extensions are incorporated on a client by navigating to a particular network or Internet site through which the extensions can be accessed. The files describing the extension files are downloaded on the client. These files tell the client where, when and how the particular extension can be plugged into the program or platform, as well as where to find the appropriate extension files and how to download them. The extension files are then downloaded and incorporated into the platform.

In one embodiment, an inventive software architecture is provided for handling and consolidating particular types of descriptive files associated with various extensions. A filtering mechanism called attachment points, is used to create handlers for the different descriptive types of files that define a software extension. Each of these handlers is known as an attachment manager. Attachment managers are provided for each extensible feature type. The attachment managers interpret data from extensions files which are supplied by attachment points. In addition to predefined attachment managers, custom attachment managers can be created using data from attachment points. When an extension extends a particular feature type, the attachment points ensure that only

1 the appropriate attachment manager is notified so that the feature type can be  
2 incorporated into the program or platform efficiently.

### 3 4 **BRIEF DESCRIPTION OF THE DRAWINGS**

5 Fig. 1 is a high level view of a system that can be utilized in accordance  
6 with one described embodiment.

7 Fig. 2 is an exemplary computer system that can be utilized in accordance  
8 with the described embodiment.

9 Fig. 3 is a diagram of an exemplary Extension Definition File (EDF) and  
10 package manifest (PKG) in accordance with one described embodiment.

11 Fig. 4 shows a portion of an EDF schema in accordance with the described  
12 embodiment.

13 Fig. 5 shows a portion of an EDF schema in accordance with the described  
14 embodiment.

15 Fig. 6 shows a portion of a PKG in accordance with the described  
16 embodiment.

17 Fig. 7 is a block diagram illustrating how file hashes can be used for  
18 versioning in accordance with one embodiment.

19 Fig. 8 is a block diagram that illustrates two exemplary package objects in  
20 accordance with one embodiment.

21 Fig. 9 is a flow diagram that describes steps in a method in accordance with  
22 one described embodiment.

23 Fig. 10 is a flow diagram that describes steps in a method in accordance  
24 with one described embodiment.  
25

1 Fig. 11 is a block diagram that illustrates an exemplary package manifest  
2 creation tool in accordance with one described embodiment.

3 Fig. 12 is a flow diagram that describes steps in a method in accordance  
4 with one described embodiment.

5 Fig. 13 is a flow diagram that describes steps in a method in accordance  
6 with one described embodiment.

7 Fig. 14 is a flow diagram of steps in a method in accordance with the  
8 described embodiment.

9 Fig. 15 is a flow diagram of steps in a method in accordance with the  
10 described embodiment.

11 Fig. 16 shows a portion of an exemplary catalog structure in accordance  
12 with the described embodiment.

13 Fig. 17 is a block diagram of a software architecture in accordance with the  
14 described embodiment.

15 Fig. 18 is a flow diagram of steps in a method in accordance with the  
16 described embodiment.

17 Fig. 19 is a diagram that illustrates one aspect of attachment point  
18 architecture in accordance with the described embodiment.

19 Fig. 20 is a diagram that illustrates one aspect of the Fig. 17 architecture.  
20

## 21 **DETAILED DESCRIPTION**

### 22 **Overview**

23 The methods and systems described just below provide a mechanism by  
24 which software can be delivered over a network, such as the Internet. In one  
25 specific example, various functionalities can be added dynamically to an

Extensions can be described utilizing a hierarchical tag-based language which facilitates handling and use of the various extensions. In one particular implementation, a software platform is provided that can incorporate various functionalities. The software platform and the inventive architecture described below enable third and fourth party developers to develop extensions for the platform that can be easily and seamlessly incorporated into the platform, while relieving the developers of any requirements associated with knowledge about how the extensions will be incorporated into the platform. Thus, the incorporation of third party extensions is essentially a transparent process, as far as developers are concerned.

Consider for example, Fig. 1 which shows a user's computer 100 and several so-called extension sources 102, 104, and 106. The extension sources can comprise any entity from which a software extension can be obtained via a network. In an exemplary implementation, the network can comprise the Internet, although other networks (e.g. LANs and WANs) can certainly be utilized. Extension sources can include, without limitation, business entities such as retail stores that might maintain a network site. In one implementation, a user can execute software on their computer that provides an application program or software platform. In this document, the terms "application program" and

“software platform” will be used interchangeably. Each of the different extension sources 102-106 can provide software extensions that can plug into the software platform that is executing on the user’s machine. These extensions are deliverable via a network such as the Internet, and assist in providing applications that can be executed on the user’s machine. In the described embodiment, the extensions are logically described in XML which is in line with emerging industry standards. Additionally, the use of XML assists in the future discoverability of extensions by promoting XML DOM properties on the Internet. It will be appreciated, however, that any suitable format can be used for describing the extensions, e.g. a binary description could be used.

Extensions can be delivered from any number of different extension sources. The inventive methods and systems provide a streamlined and organized way to handle the provided extensions. The use of XML advantageously enables efficient handling of extensions from multiple different extension sources, without unduly taxing the software components that utilize specific portions of an extension or extensions.

In one particular implementation, the software platform on the user's machine provides various different integrated functionalities that enable a user to accomplish different document-centric tasks. An exemplary system is described in the U.S. Patent Application entitled "Single Window Navigation Methods and Systems", incorporated by reference above.

### Exemplary Computer Environment

Fig. 2 shows an exemplary computer system that can be used to implement the embodiments described herein. Computer 130 includes one or more



processors or processing units 132, a system memory 134, and a bus 136 that couples various system components including the system memory 134 to processors 132. The bus 136 represents one or more of any of several types of bus structures, including a memory bus or memory controller, a peripheral bus, an accelerated graphics port, and a processor or local bus using any of a variety of bus architectures. The system memory 134 includes read only memory (ROM) 138 and random access memory (RAM) 140. A basic input/output system (BIOS) 142, containing the basic routines that help to transfer information between elements within computer 130, such as during start-up, is stored in ROM 138.

Computer 130 further includes a hard disk drive 144 for reading from and writing to a hard disk (not shown), a magnetic disk drive 146 for reading from and writing to a removable magnetic disk 148, and an optical disk drive 150 for reading from or writing to a removable optical disk 152 such as a CD ROM or other optical media. The hard disk drive 144, magnetic disk drive 146, and optical disk drive 150 are connected to the bus 136 by an SCSI interface 154 or some other appropriate interface. The drives and their associated computer-readable media provide nonvolatile storage of computer-readable instructions, data structures, program modules and other data for computer 130. Although the exemplary environment described herein employs a hard disk, a removable magnetic disk 148 and a removable optical disk 152, it should be appreciated by those skilled in the art that other types of computer-readable media which can store data that is accessible by a computer, such as magnetic cassettes, flash memory cards, digital video disks, random access memories (RAMs), read only memories (ROMs), and the like, may also be used in the exemplary operating environment.

1       A number of program modules may be stored on the hard disk 144,  
2 magnetic disk 148, optical disk 152, ROM 138, or RAM 140, including an  
3 operating system 158, one or more application programs 160, other program  
4 modules 162, and program data 164. A user may enter commands and  
5 information into computer 130 through input devices such as a keyboard 166 and a  
6 pointing device 168. Other input devices (not shown) may include a microphone,  
7 joystick, game pad, satellite dish, scanner, or the like. These and other input  
8 devices are connected to the processing unit 132 through an interface 170 that is  
9 coupled to the bus 136. A monitor 172 or other type of display device is also  
10 connected to the bus 136 via an interface, such as a video adapter 174. In addition  
11 to the monitor, personal computers typically include other peripheral output  
12 devices (not shown) such as speakers and printers.

13       Computer 130 commonly operates in a networked environment using  
14 logical connections to one or more remote computers, such as a remote computer  
15 176. The remote computer 176 may be another personal computer, a server, a  
16 router, a network PC, a peer device or other common network node, and typically  
17 includes many or all of the elements described above relative to computer 130,  
18 although only a memory storage device 178 has been illustrated in Fig. 2. The  
19 logical connections depicted in Fig. 2 include a local area network (LAN) 180 and  
20 a wide area network (WAN) 182. Such networking environments are  
21 commonplace in offices, enterprise-wide computer networks, intranets, and the  
22 Internet.

23       When used in a LAN networking environment, computer 130 is connected  
24 to the local network 180 through a network interface or adapter 184. When used  
25 in a WAN networking environment, computer 130 typically includes a modem 186

1 or other means for establishing communications over the wide area network 182,  
2 such as the Internet. The modem 186, which may be internal or external, is  
3 connected to the bus 136 via a serial port interface 156. In a networked  
4 environment, program modules depicted relative to the personal computer 130, or  
5 portions thereof, may be stored in the remote memory storage device. It will be  
6 appreciated that the network connections shown are exemplary and other means of  
7 establishing a communications link between the computers may be used.

8 Generally, the data processors of computer 130 are programmed by means  
9 of instructions stored at different times in the various computer-readable storage  
10 media of the computer. Programs and operating systems are typically distributed,  
11 for example, on floppy disks or CD-ROMs. From there, they are installed or  
12 loaded into the secondary memory of a computer. At execution, they are loaded at  
13 least partially into the computer's primary electronic memory. The invention  
14 described herein includes these and other various types of computer-readable  
15 storage media when such media contain instructions or programs for implementing  
16 the steps described below in conjunction with a microprocessor or other data  
17 processor. The invention also includes the computer itself when programmed  
18 according to the methods and techniques described below.

19 For purposes of illustration, programs and other executable program  
20 components such as the operating system are illustrated herein as discrete blocks,  
21 although it is recognized that such programs and components reside at various  
22 times in different storage components of the computer, and are executed by the  
23 data processor(s) of the computer.  
24  
25

## Extensions

An “extension”, as used in this document, will be considered to include, without limitation, software functionality and content that can be added to an application program or software platform. These additions typically provide some type of functionality that the application program may not have had before the extension was incorporated, or alter the behavior of at least one existing feature. The extension is incorporated or integrated directly into the application program in a way that changes, to some degree, the manner in which the application program behaves or operates. Extensions provide dynamically added content and can provide applications (such as an email application), plug-ins to extend existing applications (like a fax plug-in to an email application), or simply web pages, to name just a few.

In the described embodiment, extensions are described using XML, an industry-standard, text-based markup language. XML greatly facilitates the extensibility of software content. Specifically, various extensions can be authored by third parties and described in XML in a manner that enables the extensions to be readily integrated into application programs. It should be appreciated, however, the XML constitutes but one exemplary way of describing and using the extensions. Other ways can, of course, be used.

### Exemplary Extension Organization

In the described embodiment, extensions are organized in three separate but related portions: an Extension Definition File (EDF), a Package Manifest (PKG), and the code, components, or “bits” that make up or define the extension. An EDF can be, but need not be associated with a URL (Universal Resource Locator) that

provides a way for a client to access the EDF. By convention and choice, the PKG file is located at the same URL as the EDF. It will be appreciated that the described EDFs and PKG are each not required for the other to be used. It just so happens that, in the example that is given in this document, the two are employed together. To that end, each of these features can be separately and independently employed.

EDFs describe logical attachments to an application program or software platform, while PKGs specify the physical files and resources that are used in an extension. There can be a one to one correspondence between EDFs and PKGs.

Fig. 3 shows an exemplary organization 300 that includes an EDF 302 and a corresponding package manifest (PKG) 304. In the illustrated example, the EDF 302 uses XML, a hierarchical tag-based language, to describe the logical attachments or extensions to an application program. The corresponding PKG 304 also specifies the physical files and resources that are associated with a particular extension in XML. Exemplary file types are shown to the right of PKG 304 and include, without limitation, HTM, GIF, UDO, XML, DLL, and various other types of files.

### Extension Definition File (EDF)

In the described example, an EDF is an XML file that logically describes an extension. For example, the EDF can describe HTML that makes up a user interface (UI), the objects that contain code for implementing various functions, and the like. The EDF can also contain all or part of the functionality that comprises an extension. For instance, the HTML that describes a toolbar could be incorporated directly into an EDF file, and a toolbar attachment manager could

1 read it from the EDF file, instead of from a URL. The information contained in  
2 the EDF is processed and (together with the information contained in the PKG),  
3 the appropriate files are automatically installed on a user's computer. This is done  
4 unobtrusively without manipulating the computer's persisted settings, as might be  
5 found in the user's system registry.

6 An EDF, implemented in XML, contains various tags that are associated  
7 with various extensions. The various tags can correspond to:

- 8 • User interface elements
- 9 • Behaviors/Components/Objects
- 10 • Store Elements
- 11 • User-defined objects
- 12 • Or anything else that represents a point of extensibility in the  
application or platform

13 EDFs advantageously have an "open schema" which means that third party  
14 developers can extend the extension mechanism and include their own extensions  
15 by creating their own tags. Additionally, extensions can themselves be extended  
16 by other developers. EDFs can also have one or more predefined tags. Exemplary  
17 predefined XML tags for user interface elements can include tags for feature types  
18 such as: tool bars, accelerators, menu items, and themes. These feature types are  
19 utilized in the single navigable window application incorporated by reference  
20 above and defined in the table immediately below:

Feature Type	Definition
Tool Bars	Horizontal command containers above the document area.
Accelerators	Keyboard shortcuts for commands
Menu Items	Pop-up or drop-down menu choices that third parties can add to well-known, named menu attachments in the platform
Themes	A data-driven way to provide overrides for well-known resources of the platform, such as default buttons or default style sheet

Table 1

Exemplary predefined XML tags for behaviors/components/objects include tags for Services. These feature types are utilized in the single navigable window application incorporated by reference above and defined in the table immediately below:

Feature Type	Definition
Services	Services are objects that extend existing objects (such as the application, window, or document) in the application or platform's Object Model. For example, editing functions use Object Model Attachments attached to the window or document that maintain document context and editing state per-window. These can also include Object Model Attachments attached to the application (such as a spellchecker dictionary object)

Table 2

Exemplary predefined XML tags for store elements include tags for content classes and offline data sources. These feature types are utilized in the single navigable window application incorporated by reference above and defined in the table immediately below:

Feature Type	Definition
Content Classes	Allow extension writers to define new types of XML documents with new schemas.
Offline Data Sources	Allow for extension writers to define store replication instructions in an EDF.

Table 3

### EDF Schema

In the described embodiment, the EDFs have a particular XML schema that is utilized. The schema comprises collections of XML tags that are arranged in a hierarchical organization to facilitate information dissemination to software components that need certain extensions. In the described embodiment, the outer (encompassing tag) for EDFs is an “extension” tag.

Fig. 4 shows an exemplary extension tag. “extension” tags can include one or more of the following attributes, all of which are optional:

Attribute	Definition
urn	ID for the extension. It allows extension writers to specify relative locations for content in EDFs without using relative paths or fixed URLs. It also allows hosting administrators to move around extensions on servers without breaking any links.
name	Name that can be used in a status bar or message display
version	Vendor-determined version number for the extension.
lastUpdate	Date/time that the EDF was last modified.
description	Brief description of the extension.

Table 4

Within the “extension” outer tag are one or more child tags, also referred to as “top level tags”. These top level tags are each associated with a feature type



that can be added by a particular extension. Exemplary feature types are discussed in connection with Tables 1-3 above. Underneath each top level tag there can be one or more child tags that are individually associated with a particular feature of the feature type that is to be added by a particular extension.

Fig. 5 shows an exemplary XML schema organization in accordance with this embodiment. For each top level tag in an EDF, there is an associated attachment manager which is a software component that receives data associated with the tag so that the data can be used to incorporate the extension into the platform or application program. Different attachment managers may interpret the data from the tag in different ways to provide different types of extensibility, so different top level tags will contain different types of data in different structures. This will become more evident in the "Architecture" section below. Note that the "edf:" XML namespace qualifier enables support of an open schema where extensions can provide their own tags and corresponding attachment managers. Tags within the edf namespace are used by built-in attachment managers in the application or software platform. Tags in other namespaces are used by third-parties to provide additional points of extensibility.

### **Package Manifest (PKG file)**

The package manifests (PKGs) assist in organizing the downloading of software in the form of multiple files over a network such as the Internet. The PKGs are advantageously employed, in the example given in this document, with EDFs. As pointed out above, however, the techniques discussed in connection with the PKGs can be deployed independently of EDFs and in any suitable scenario where it is desirable to deliver software over a network such as the

In designing a delivery mechanism for Web-assisted delivery of software content or files, several considerations are of interest.

Whenever possible, it is desirable to reduce the size of required downloads during update and installation operations. To address this consideration, software content is broken into multiple packages. Each package contains a group of one or more files that implement a common or well-defined functionality. By breaking the content into individual packages, the size of the required download during installation and update can be minimized. Each package is then described by a package manifest (PKG) that includes file information such as file locations and hash values that can be used for validation or security and versioning.

It is also desirable to give the end user a Web-like experience. To do this, extensions are loaded in a manner that feels to a user more like they are loading a web page, rather than traditional software packages where the user has to wait until the entire package is loaded before they can interact with it. In the described embodiment, users are given a web-like experience by streaming the extension files down to the client so that a user can begin to interact with an application program much sooner than if they had to wait for the entire software application program to load. For example, if there are user interface (UI) image files streaming down, the user can see the UI as the files stream in. Consider, for example, the single application program having the multiple different functionalities that is described in the patent application incorporated by reference

1 above. A user might browse to an email functionality and download the files that  
2 are necessary to interact with the email functionality. Files that are associated  
3 with another different functionality would then be downloaded after the files  
4 associated with the email functionality. In this way, the user can begin to operate  
5 within a particular functionality without having to wait for all of the files  
6 associated with all of the other functionalities.

7 Another consideration of interest pertains to the efficiency with which the  
8 extension files or “bits” are delivered to the client. To address this consideration,  
9 the described embodiment utilizes a couple of different download approaches: a  
10 throttled download and a background download. Throttled downloading conducts  
11 download operations while taking into account the available bandwidth and type  
12 of media over which the files are being transferred. Any suitable throttled  
13 download process can be employed and will be understood by those of skill in the  
14 art. Background download is conducted while the user is working within the  
15 application program and is implemented by allocating a background thread so that  
16 the user can continue their work. One optimization that is utilized is that packages  
17 are prioritized and delivered in accordance with what a user might be working on.

18 Another consideration is associated with optimizing the user’s computing  
19 experience. Here, the user’s experience is optimized by making available the most  
20 common scenarios for the user. This is effectively directed to giving the user the  
21 functionality that they want first, and then, through the background download  
22 process, providing the code that implements functionalities that the user might use  
23 in the future. To determine which functionalities a user desires to have first, an  
24 automated scenario-based packaging process is provided that runs against file  
25 usage logs from scripted scenarios.

All of these considerations and the inventive solutions directed to addressing the considerations are discussed in detail in the sections that follow below.

### Package Manifest Definition

In the described embodiment, a package manifest (PKG file) comprises a list of files that are utilized in a package. The list is advantageously compressed somewhat and digitally signed. Each package manifest can contain a list of one or more files each of which can include an associated hash, as well as download directives that control caching of the files. Once an extension is authored, a software tool can be used to generate the package manifest.

In addition, the package manifest can specify several other pieces of information:

- **FILE GROUP**

All files in an extension can be labeled according to a number of predefined file groups. The file group of a particular file determines when the particular file gets downloaded, where it is stored on the client, and how it gets packaged. In the described embodiment, four predefined file groups are provided and are listed and described in the table immediately below:

Group name	When downloaded	Where stored on the client	Packaging	Content
Required	Downloaded before any other files in the extension.	NetDocs package cache	All required files in an extension are	DLLs included so that a user will not have to wait for a prolonged

			packaged together as a CAB* file.	period of time before clicking on a UI element
Offline	Offline files start getting downloaded as soon as Required are down. Providing the user stays on line long enough, these files will all get downloaded and will later be available for offline use.	NetDocs package cache	File are sent down individually.	Bulk of the UI files.
On demand	Only downloaded when they are requested for the first time.	NetDocs package cache	Files are sent down individually.	To avoid using up disk space on the client, advanced features can be put in this category.
Online only	Downloaded on demand. Content is only available when the user is online.	WinInet Cache	Files are sent down individually	Content that is not to be provided offline. Examples include help pages and other content that can consume a large amount of disk space.

\* CAB stands for the CABinet technology that Internet Explorer uses to package bits for download. CAB files average from 2 to 3:1 compression, and are optimized for quick expansion. CAB files have the added security benefit that they are easily signed.

## • FILE DOWNLOAD PRIORITY

Files in each group are listed according to the order in which they should be downloaded. This download order is implicit in the ordering of the files in the package manifest, an example of which is shown in Fig. 6.

## • HASH VALUE FOR SECURITY/VERSIONING

Individual files in the package manifest can have an associated hash value. Each hash value is generated by running the file through an encryption algorithm. An exemplary encryption algorithm is Microsoft's CryptoAPI. In the illustrated example, each file can be listed with a base 64-encoded hash value, so that the file can be validated once the content arrives at the client. Specifically the package manifest is sent to the client in a secure manner (i.e. it is digitally signed). The

1 package manifest contains the hash values for individual files. When the  
2 individual files are received by the client, each of the files can be run through the  
3 same Crypto API that was used to provide the hash values in the package  
4 manifest. If the hash values for a particular file compare favorably, then the file  
5 has not been altered and is secure.

6 When a file is updated, hash values can serve a useful purpose in  
7 identifying files that have not been changed between different versions of an  
8 extension. Consider Fig. 7, for example. There, an old directory 700 in a client  
9 package cache contains package A which include two files—file 1 with hash = x,  
10 and file 2 with hash = y. Assume that this package is associated with an older  
11 version of an extension. When an updated version is produced, its package  
12 manifest is delivered to the client. The updated extension version is represented  
13 on a source directory of a code or web server 704. The package manifest includes  
14 the hash values for all of the files in the new extension version. A new client  
15 destination directory 702 is defined for all of the files of the new extension. If any  
16 of the hash values for files in the new extension are the same as the hash values of  
17 the files in the old directory 700, then those files can be copied directly from the  
18 old directory 700 to the new destination directory 702. In this example, file 1's  
19 hash value is the same as the hash value for file 1 on the source directory 704, so it  
20 can be copied into the new destination directory 702. File 2's hash value, however  
21 is different from the hash value for file 2 on the source directory, so it is not copied  
22 from the old directory 700. Rather, file 2 is downloaded from the code server. A  
23 new file 3 has been added and is also downloaded from the code server. Hence, in  
24 this example, a new version of an extension resulted in a download of less than all  
25 of the files in the extension version. This is because hash values for each of the

Using hash values for versioning has two important advantages over traditional versioning schemes. First, the update process is automatic. That is, with an explicit version number, it is possible to forget to update the version number when shipping a new release of a file. Using hash values avoids this problem. Second, versioning does not rely on file types. Specifically, traditional versioning schemes commonly embed version information within files; however, not all files (e.g. GIF files) support embedded version information. In the present example, using hash values for versioning does not depend on whether a particular file type supports or does not support embedded version information. In addition, the version information can be stored separately from the file itself. Thus, access to actual file to determine whether it is current is not needed.

- TOTAL STORAGE SIZE OF PACKAGE

The total storage size of a package is useful at download time to verify that the user has sufficient disk space.

- ClassID's FOR THE DLLs

Listing 'ClassIDs for each DLL is necessary to enable script writers to create classes by scripting against the OM. Additionally, this enables a determination of which package contains the code for a particular class.

## DLL LOAD DEPENDENCIES





1 explicitly requested a file/extension by clicking, for example, on an extension link,  
2 or requested an action, for example, by clicking on the “Compose” mail button,  
3 that requires download of files which are not available locally.

4 Along with background downloads, a queue management feature is  
5 provided. Specifically, when an extension must be installed or updated, a package  
6 manager, which is essentially a software component that manages packages, is  
7 provided with the following information:

- 8 • URL for the package manifest information on a code server
- 9 • URN for package destination directory in the package cache at the client
- 10 • (Optional) URN for the old package directory (if one exists) in the  
11 package cache

12 From this information, the package manager creates a package object and  
13 adds the package object to a download queue. The download queue is designed  
14 for easy rearrangement of a package download order. Consider, for example, Fig.  
15 8 which shows a portion of a download queue 800 that contains two package  
16 objects—package object 802 (corresponding to package A) and package object  
17 804 (corresponding to package B). The package objects maintain a list of which  
18 files of a corresponding package have been downloaded or installed. In the  
19 present example, files 1 and 2 from package A have been installed while file 3 has  
20 not been installed; and files 1, 2, and 3 have not been installed from package B.  
21 The download queue can be rearranged based on what the user is doing. That is,  
22 based on the actions that a user takes, the priority of files that are to be  
23 downloaded can be changed. In this example, the package manager is designed to  
24 process the first uninstalled file in the package at the head of the download queue.  
25 If, however, the user starts to use a file in an extension that is different from the

1 extension whose files are at the head of the download queue, the corresponding  
2 package for the file that the user has started to use can be moved to the head of the  
3 download queue. Because a file's package is specified by its URN, the file's  
4 package can be quickly identified and located in the download queue. For  
5 example, and considering Fig. 8, if one of the files in package B is requested  
6 before the package manager has started to install package A's third file, then  
7 package B will be moved to the head of the download queue.

8 Fig. 9 is a flow diagram that describes steps in a download queue  
9 management method in accordance with the described example. The method can  
10 be implemented in any suitable hardware, software, firmware or combination  
11 thereof. In the present example, the method is implemented in software.

12 Step 900 receives one or more requests for an extension. The requests can  
13 be generated in any suitable way. Step 902 creates a package object that  
14 corresponds to each extension package that is to be downloaded. Step 904  
15 arranges the package objects in a download queue. Step 906 then downloads files  
16 corresponding to the package objects in the download queue. This step can be  
17 implemented by, for example, starting at the head of the download queue and  
18 downloading files until all of the files for a package object have been downloaded,  
19 and then moving to the next package object. Step 908 ascertains whether a user  
20 action requires a file that is not described in the current package object. If the  
21 user's action does not require a file not described by the current package object,  
22 then the method branches back to step 906 and continues to download files  
23 associated with the current package object. If, on the other hand, the user's action  
24 requires a files that is not described in the current package object, then step 910  
25 moves the package object associated with the required file to the head of the

1 download queue and begins to download files associated with this newly-  
2 repositioned package object. This step can be implemented by ascertaining which  
3 package object is associated with the required file by ascertaining the URN  
4 associated with the file. This URN specifies the file's package so that its package  
5 object can be quickly located and moved to the front of the download queue.

### 6 7 Package Creation

8 One of the innovative features of the described embodiment is its  
9 extensibility. That is, a software platform is provided in the form of an application  
10 program that can be extended by various third-party user-defined extensions.  
11 These extensions are delivered via the Web and are integrated directly into the  
12 software platform. In order to provide an organized delivery process, packages  
13 should be created in a uniform manner so that they can be predictably handled and  
14 integrated into the software platform.

15 In accordance with the described embodiment, each package should  
16 correspond to an end-user feature. For example, in the patent application  
17 incorporated by reference above, separate packages are provided for each of the  
18 email, contacts, document authoring, and planner functionalities. If packages that  
19 do not depend on one another share a dependency, then this shared dependency  
20 should become its own package. For example, there is no reason why the email  
21 and document authoring functionalities should depend on one another, yet both of  
22 them require the ability to publish content. By separating the publishing  
23 functionality into its own package, a certain amount of download order flexibility  
24 is preserved. Depending on what the user starts to work on, the files  
25

1 corresponding to the email functionality or the document authoring can be  
2 downloaded first.

3 Fig. 10 is a flow diagram that describes steps in a package creation method  
4 in accordance with the described example. The method can be implemented in  
5 any suitable hardware, software, firmware or combination thereof. Portions of the  
6 method might, however, be implemented manually.

7 Step 1000 identifies end user features that are to be provided as extensions.  
8 Step 1002 identifies any shared dependencies among the end user features. Step  
9 1004 creates individual packages for the end user features. Step 1006 creates  
10 individual packages for any shared dependencies among the end user features.

### 11 12 Automated Package Manifest Creation Tool

13 Advantageously, and in accordance with one implementation, an automated  
14 package manifest tool is provided and takes various input parameters and  
15 automatically creates a package manifest. The tool can be available to third  
16 parties to assist them in creating a package manifest.

17 Fig. 11 shows an exemplary package manifest creation tool 1100 that is  
18 desirably implemented in software. In this specific example, the tool can take the  
19 following input parameters (some of which are optional):

- 20 • Extension directory
- 21 • File group information and DLL load dependencies (Optional)
- 22 • File usage statistics from scenario runs (Optional)

23 The extension directory input parameter specifies the directory containing  
24 all of the files that will be described by the package manifest. If this is the only  
25

parameter, then tool 1100 will generate a manifest in which the EDF and DLLs in the directory are listed in the “Required” set, and all other content is “Offline”.

The file group information and load dependencies parameter is optional. If an extension author has an idea of the categories in which his or her files should be placed, the categories should be specified here. For example, the author of the template manifest shown below knows that he wants his error handling GIF to be included in the package’s required set. His choices here will always be respected in the final manifest. Additionally, if the extension author knows of any DLL load dependencies, these should be specified here as well.

```
<?xml version="1.0"?>
<Package>
  <Files>
    <Group Name= "required">
      <File Name = "bar.dll"/>
      <File Name = "foo.dll"/>
      <Dependencies>
        <File Name= "bar.dll"/>
      </Dependencies>
    </File>
    <File Name= "myextension.edf"/>
    <File Name= "errorhandling.gif">
  </Group> ...
</Files>
</Package>
```

The file usage statistics from scenario runs parameter is an optional parameter. This parameter enables the file download priority to be determined based on scenario runs. A scenario is a script of tasks that the average user typically follows when using a product during a particular portion of product use. For example, one scenario might pertain to the tasks involved in sending an email

1 message (i.e. click "new mail" button, type in "TO" well, type is "Subject" well,  
2 etc.). In the described embodiment, file usage statistics from scenario runs are  
3 collected from running IIS logs on various scenarios. The different scenarios are  
4 directed to ensuring, with some degree of probabilistic support, that the file  
5 download order reflects, in some way, the files that will likely be used by the user  
6 first.

7 It will be appreciated that the file usage statistics can be provided  
8 dynamically by building a knowledge base that describes the actual tasks that  
9 people typically accomplish. The information that is maintained in the knowledge  
10 base can then be used to generate and adapt download scenarios that actually  
11 conform to patterns that are established across a user base.

12 If extension writers want to pass the package manifest creation tool 1100  
13 this information, they need to specify the log directory, as well as the start and end  
14 dates of the section of the log that the tool should analyze. For third parties, the  
15 download priority order within a group will be the order in which the group's files  
16 were requested in the logs across all scenarios.

17 In one implementation, the approach is somewhat more sophisticated.  
18 Additional information (in addition to the scripted steps) is stored in the IIS logs  
19 and includes scenario priority and checkpoints. The scenario priority is a priority  
20 that is assigned for each scenario. So, for example, if one scenario is ten times  
21 more important than another scenario, this information can be maintained. The  
22 priority (e.g. a rating from between 1 to 100, with 100 being the highest priority),  
23 should be equal to a best guess as to the percentage of the time that users will step  
24 through the scenario, assuming they use the extension at all. Checkpoints provide  
25 a way to separate one scenario from another. For example, checkpoints designated

as "Offline" and "Shutdown" can be automatically added at the beginning and end of the scenarios, respectively, so there can be a differentiation between scenario runs in the log. Additionally, script authors can optionally use checkpoints mid-scenario to indicate a change in group priority, e.g. part of a scenario script could be labeled as an "On demand" feature and another part could be labeled as "Offline".

Fig. 12 is a flow diagram that describes steps in a package manifest creation method in accordance with the described example. The method can be implemented in any suitable hardware, software, firmware or combination thereof. In the described example, various steps of the method are implemented by a manifest creation tool that is implemented in software.

Step 1200 provides a package manifest creation tool. This tool can be a software tool that resides on an extension author's machine. Step 1202 receives information pertaining to an extension directory as a first input parameter. Step 1204 determines whether there is any file group information or load dependency information provided by the extension author. If there is, step 1206 receives the information as an input parameter. Step 1208 determines whether there is any file usage statistic information. Such information can be provided, in one embodiment, through the use of scenario runs as described above. If such information is provided, then step 1210 receives the information as an input parameter. Step 1212 then uses all of the information provided as input parameters to generate a manifest automatically.

## Exemplary File Ordering Heuristics Based on File Usage Statistics

Fig. 13 is a flow diagram that describes but one exemplary file ordering or sorting heuristic in accordance with the described embodiment. It is to be understood that this specific example constitutes but one way of ordering files for download. Accordingly, other heuristics can be used without departing from the spirit and scope of the claimed subject matter.

Step 1300 sorts files by file group. Recall that in the illustrated example above, files can be grouped in one of four possible groups: Required, Offline, On Demand and Online Only. A file's group is determined first by the manifest, and, if it does not provide any group information, then by the highest priority group that it uses, according to checkpoint information in the log. Files in the "Required" set should not be considered because their order is already known. If no group information is included about a file, then an assumption is made that the EDF and all DLLs are "Required" files and all other files in the directory are "Offline".

Consider, for example, the following initial file usage information for three different scenarios:

Scenario 1 file usage: 1) FileA.gif, 2)FileB.xml, 3)FileE.dll

Scenario 2 file usage: 1) FileC.xml, 2) FileA.gif

Scenario 3 file usage: 1)File D.js, 2)FileA.gif

Scenario 1 = priority 80

Scenario 2 = priority 80

Scenario 3 = priority 40

In this example, there are three scenarios that have files associated with them. Each of the scenarios has a priority with which it is associated. The files



are first sorted by group (step 1300). Recall that in this ordering heuristic, DLLs are "Required" and all other files are considered "Offline". This provides the following sorted files:

Required files  
FileE

Offline files  
FileA, FileB, FileC, File D

Step 1302 sorts files based on scenario priorities (from highest to lowest). Higher priority files are ordered so that they are downloaded first. This step provides the following sorted files:

Required files  
FileE

Offline files  
Priority 80 group: files used by Scenarios 1 & 2 = File A, File B, and File C

Priority 40 group: files used by Scenario 3 (that are not already listed) = File D.

Step 1304 then sorts the files by file usage order within a scenario run. For each priority grouping with more than one file, the files are sorted according to the average order in which they were downloaded within scenarios of their labeled priority. Scenarios with a smaller average usage order will be downloaded earlier. Ties are broken based on the order in which the scenarios appear in the input file. As an example, consider the following:

File A: average order = (Scenario 1 order + Scenario 2 order)/2 = (1+2)/2 = 1.5.

File B: average order = (Scenario 1 order)/1 = (2)/1 = 2.

File C: average order = (Scenario 2 order)/1 = (1)/1 = 1.

Here, file A got used first by scenario 1 and second by scenario 2 for an average of 1.5, and so on. File C has the smallest order number so, of the Offline files, it is sent first. The final file order is shown below:

Required files

FileE

Offline files

FileC, FileA, FileB, File D

**Code, Components and “bits”**

The following files and resources can be, but need not be included with an extension. This list is not exclusive, as other resources can certainly be incorporated into an extension.

- Customized UI and keyboard shortcuts
- Components and Behaviors
- XML browsing and editing components (including XSL and business logic objects)
- Static pages or other resources
- Third-party defined custom content

Users install extensions by navigating to a network site for the extension. In an Internet implementation, the user navigates to an appropriate URL for the extension. Hosting administrators can also “push” extensions so that users can

1 automatically receive them by adding an entry into the appropriate users'  
2 "Preference" settings.

### 3 4 **Platform Set Up and Extension Installation**

5 Fig. 14 is a flow diagram that describes exemplary steps in a set up and  
6 extension installation process in accordance with the described embodiment. This  
7 example describes an Internet-based example. In the illustrated example, various  
8 extensions are maintained at or accessible through various Internet sites. The  
9 extensions are deliverable via the Internet to a client. It will be appreciated that  
10 the illustrated division of computers may not necessarily exist at all. For example,  
11 all of the functionality embodied by the computers may reside on one machine, or  
12 the extension may be local, or the platform and the extensions may be on the same  
13 machine, etc.

14 The flow diagram, in this example, is illustrated in connection with three  
15 separate "zones", one of which represents a client, one of which represents a  
16 "platform" Internet server, and one of which represents a third party Internet  
17 server. The acts that are described in connection with the different zones are  
18 performed, in this example, by the entities assigned to the zone. In some  
19 configurations, one or more of these zones may overlap. For instance, the  
20 platform server may be the same device as the extension server.

21 Step 1400 navigates a user to a particular Internet site that is associated  
22 with the software platform that is to be utilized as the foundation for extension  
23 installation described below. In step 1402, the user clicks an "install" button that  
24 sends a message to the software platform server that indicates that a user wishes to  
25 install the software platform. This step can be an optional step. Steps 1404 and

The steps described immediately above constitute steps associated with an initial set up in which the software code for the single navigable window application is delivered to and installed on a client machine. The steps described immediately below are associated with extension installation.

Step 1410 uses a link that is associated with an extension to access the extension. This step can be implemented by a user navigating their browser to a particular Internet site through which one or more extensions can be accessed. Alternately, a reference to the link can be placed in the user's preferences or the preferences of a computing group with which the user is associated (e.g. the system administrator can place a reference in a group's preferences). The link can advantageously be associated with a third party Internet server or Internet site. Step 1412 downloads extension files according to the PKG associated with an EDF. The files are delivered to the client and step 1414 places the extension files in a local store as specified by the PKG specification. At this point, an extension is installed and the user can utilize the functionality provided by the extension. Step 1416 determines whether extension updates are available. This can be done by periodically polling an extension catalog (discussed in an "Extension Catalog"

1 section below) to ascertain whether there are any updates to various extensions.  
2 Alternately, notifications might be automatically sent to the client so that the client  
3 is aware of updates or any other method might be used to determine if updates are  
4 available. If there are updates available, step 1418 branches to step 1412 which  
5 downloads the extension files associated with the update and installs them on the  
6 client.

### 7 8 **Developing Extensions**

9 Developing extensions for the software platform is a fairly straight-forward  
10 process. A developer develops the extension content using a tool such as Notepad  
11 or other tools such as Visual Studio. The extension is then described in an EDF  
12 and PKG and the PKG is digitally-signed and then optionally compressed. The  
13 extension can then be hosted on a particular network server.

14 Fig. 15 is a flow diagram that describes steps in an extension development  
15 method in accordance with the described embodiment. One or more of these steps  
16 can be performed by a software developer or organization that creates a particular  
17 extension. Some of the steps are implemented in software. Step 1500 develops an  
18 extension. Any suitable tools can be used to develop the extension. Step 1502  
19 then creates an extension definition file (EDF) for the extension. The EDF, in this  
20 example, is defined using XML as discussed above. Other formats can, of course,  
21 be used to describe the EDF. Step 1504 creates a package manifest (PKG) for the  
22 extension. The PKG, in this example, is defined using XML as discussed above.  
23 Step 1506 then hosts the EDF and PKG on a network server such as an Internet  
24 server. Additionally, the associated extension files that are described in the PKG  
25 can also be hosted by the network or Internet server (step 1508). Having

1 accomplished the above, users can now navigate to an EDF directly (using, for  
2 example, the associated URL or some other network address), which then installs  
3 the extension by caching any required files locally and placing a reference to the  
4 extension in the user's preferences.

5 Specifically, step 1510 delivers the EDF and PKG files to a client. This  
6 step can be implemented by a user navigating to a particular Internet site where the  
7 appropriate files are hosted and then downloading the files. Step 1512 delivers the  
8 extension files that are associated with the EDF and PKG files to the client,  
9 whereupon they can be installed and used.

### 10 11 **Extension Catalog**

12 One optimization, discussed briefly in connection with Fig. 14, is an  
13 extension or EDF catalog which provides an additional level of indirection to the  
14 EDF. An EDF catalog allows organizations to group extensions and provides a  
15 single place to determine when an extension changes. The desired extension can  
16 be automatically selected from the catalog by the software platform based upon  
17 the user's settings. The catalog can be queried to determine which extension is  
18 most appropriate for the user.

19 In the described embodiment, a catalog is an XML file that contains  
20 mappings from extension URNs to one or more package URNs based upon  
21 language, version or other attributes. Catalogs can, however, be defined using any  
22 suitable format. Catalogs can provide:

23  
24 The ability for a hosting organization to update version information  
25 for one or more hosted extensions in a single place

- Optional automatic indirection to correct version based upon user's settings. For instance, a catalog may list several versions of an extension for different languages. The catalog file can be processed to find the version of the extension that matches the user's language settings.
- Optional automatic upgrade to new versions of extensions as they become available

Like EDFs, catalogs can be compressed and digitally signed to prevent tampering. By subscribing to a catalog in order to detect changes for one or more hosted extensions, the number of server pings required from the client (or notifications to the client) in order to discover extension updates can be reduced.

Fig. 16 shows an exemplary XML catalog structure. Entries in the catalog can be structured as follows:

Attribute	Type	Required	Description
extensionURN	uri	Y	Identifier for an extension. There may be more than one entry for a given extension urn in a single catalog representing different versions, languages, etc.
name	String	N	Friendly name for an extension.
packageURN	uri	Y	Required urn for the package. Package urn corresponds to a discrete set of bits. It is different from the extension urn: For each user, the extension urn (name) corresponds to a specific set of files based upon language preferences and version. This means that, for shared machines, different users may have different extensionURN to packageURN maps based

			upon their preferences.
packageURL	uri	Y	url of the digitally signed compressed file containing the PKG file is required.
language	String	N	Language is an optional language specifier.
version	String	N	Version is an optional version specifier
defaultLanguage	String	N	DefaultLanguage is an optional attribute specifying the default language package. For a given version of an extension, there should be only one entry with DefaultLanguage attribute.
defaultVersion	String	N	DefaultVersion is an optional attribute specifying default version for an extension. For a given extension urn and language attribute there should be only one entry with DefaultVersion attribute.

In this particular example:

- The default language of the netdocs-planner is the English version.
- The default English version is 1.1. The default French version is 1.0. If there is no version available in the user's specified language on the platform, they will get the English version 1.1 by default.
- The English version of netdocs-planner has been upgraded from V1 to V1.1.
- There is also a French version. The extension URN is the same as the English version. There is no 1.1 release for French yet, so 1.0 is the current version for French speaking users.
- A query against the catalog returns only the rows where language matches the user's language preferences. The query would also return all rows where language is the user's language or default = 'yes' and throw out duplicates for the same name.



## Architecture

In the described embodiment, one of the aspects that provide desirable utility is the extensibility of the software platform. That is, third and fourth party developers are free to develop their own extensions which can then be used within the framework of the software platform. The extensions are integrated directly into the software so that the platform's functionality is modified by the extensions. Recall that in order to provide an extension, the developer simply authors the extension, describes their extension in an EDF and PKG, and then hosts the EDF, PKG, and associated files on a network server.

The EDF, as pointed out above, can be defined in an XML schema that includes a root node (i.e. the "extension" tag) and one or more child nodes. In this particular example, the child nodes generally correspond to the individual extension feature types that are desired for incorporation into the software platform. For example, recall that Tables 1-3 above describe various exemplary predefined feature types that can be added through an extension using the predefined XML schema.

Consider now a developer who wants to add two menus and a toolbar to the software platform. The menus and toolbar might be associated with a retail store that maintains a Web site for its customers. The retail store might want a customer who visits its Web site to be presented with a UI that is unique to the retail store and provides services that are specifically tailored to the store. To do this, the developer develops two different menus, one of which might be associated with displaying the most recent specials, and other of which might be associated with providing a search mechanism through which the user can search for specific products. The toolbar might contain specific buttons that are unique to the retail

1 store. A simplified EDF called “*retail.edf*” for the retail store’s extension is  
2 shown directly below:

```
3  
4 <edf:extension name= “retail extension” urn= “extension.retail.com”>  
5   <edf:menus>  
6     <edf:menu url= “url1.htm”/>  
7     <edf:menu url= “url2.htm”/>  
8   </edf:menus>  
9   <edf:toolbars>  
10     <edf:toolbar url= “url3.htm”/>  
11   </edf:toolbars>  
12 </edf:/extension>
```

13 Here, the outer “extension” tag designates this XML file as an extension.  
14 The inner “menus” and “toolbars” tags are top level tags that designate that the  
15 information between these tags pertains respectively to menus and toolbars that  
16 correspond to the extensions that the developer has added. The boldface “menu”  
17 and “toolbar” tags describe data pertaining to the actual extension and contain a  
18 URL that is associated with each extension as described above. The EDF above  
19 logically describes the extensions that are being provided as including two menus  
20 and one tool bar.

21 Consider also that the above EDF can constitute but one of many EDFs that  
22 are loaded into the system. Each EDF can contain one or more top level tags, each  
23 of which is associated with one or more specific extensions that are to be added to  
24 the software platform.

25 Fig. 17 is a block diagram of an exemplary software architecture that is  
configured to process multiple different EDFs so that the software components  
that are responsible for incorporating each particular extension into the software

1 platform receive the appropriate information that is specific to their extension.  
2 This example is specific to the XML implementation that is discussed throughout  
3 this document. It is to be understood that other architectures, similar in  
4 functionality to the one discussed below, can be used in other implementations  
5 without departing from the spirit and scope of the claimed subject matter.

6 Utility objects, herein referred to as attachment points, are used to process  
7 the information from the multiple EDFs. An attachment point is simply a  
8 collection of objects that fire events to registered listeners as objects are added to  
9 or removed from the collection. Many types of attachment points can be created,  
10 but all take data from a source (often another attachment point), process it (either  
11 dynamically or statically), and expose the results of their processing. Some of the  
12 simplest attachment points include:

- 13 • An XML attachment point, which loads an XML file and exposes  
the top-level nodes of the XML as objects in its collection.
- 14 • A filter attachment point, that connects to another attachment point  
and exposes only those objects from it that meet some criteria.
- 15 • A merge attachment point, that connects to one or more other  
16 attachment points and exposes all of their objects as one, merged  
collection of objects.

17 In the illustrated example, the architecture includes a collection of one or  
18 more attachment points, including a funnel structure known as an EDFHub 1700,  
19 an attachment point manager 1702, and multiple attachment managers 1704. The  
20 EDFHub 1700 receives all of the EDFs and merges them together and exposes  
21 them as a single list. Other individual attachment points provide mechanisms that  
22 manipulate (including filter, merge and expand) the single list that is exposed by  
23 the EDFHub 1700. Whenever a new extension or EDF is added to or removed  
24 from the EDFHub, the various attachment points will see to it that the appropriate  
25 attachment manager(s) is notified. This is done by firing events to the appropriate

attachment managers. The attachment point manager 1702 creates, destroys and manages the various attachment points in the system and allows easy reuse of identical attachment points.

For each top level tag (i.e. “menus” and “toolbars” tags), there is a corresponding attachment manager 1704 that uses data provided by the attachment points to incorporate a particular type of feature within the software platform. Each attachment manager requests a set of attachment points from the attachment point manager 1702. These manipulate the data exposed by the EDFHub 1700. In the illustrated example, the attachment points can be requested as a predicate chain that the attachment point manager uses to create and build a set of attachment points that operate on the data exposed by the EDFHub 1700.

Fig. 18 is a flow diagram that describes steps in a method in accordance with the described embodiment. The method is implemented in software and, in this example, by the software components of Fig. 17.

Step 1800 receives multiple EDFs. These files can be received in any suitable way. For example, a user can specify in their preferences particular extensions that they desire to be loaded when they are online. Alternately, a user might navigate using a link to a particular Internet site that recognizes that the user is executing a software platform that is configured to dynamically add the extensions. The EDFs, in this example, are funneled into the EDFHub 1700 which uses attachment points to combine the EDFs (step 1802). In this example, the EDFs are defined as XML files and the nodes are combined into one single XML list. Step 1804 exposes the combined EDFs. In this particular example, the EDFs are combined into a single XML list that is exposed to other various attachment points which then further manipulate that data (step 1806). One goal of the



powerful. This is especially true if the attachment points only process the changed data in step 2, since they only do a small amount of simple work at any one time. In the exemplary system, this incremental processing means that the whole system does not have to be re-queried when a new extension is installed or when an existing extension is removed. Additionally, each attachment manager in the exemplary system uses a specific chain of attachment points and so is only informed of changes that impact its area of extensibility.

The attachment point manager performs two important functions when building chains of attachment points. First, it allows the chains to be described as predicate strings. The attachment point manager interprets these strings and builds the required chain of attachment points. Second, it allows the reuse of identical attachment points, which makes the system more efficient. As the attachment point manager creates each chain of attachment points, it tracks which predicate string corresponds to which attachment point. If that predicate string is later requested again, it simply reuses the existing attachment point without creating a new one.

As an example, consider that the attachment manager associated with menus has requested the following predicate chain of attachment points that utilizes the *retail.edf* file above: (Note: this example does not assume the presence of an EDFHub attachment point).

```
Explode(Filter("menus", Explode(URL("retail.edf"))))
```

This string represents all of the menus in the *retail.edf* file. The XML file located at *retail.edf* is loaded by the URL attachment point which exposes the root

1 node of the XML file as the only object in its collection. The inner Explode  
2 attachment point uses the URL attachment point as its data source and exposes all  
3 of the children of the objects in that source collection. In this case, the children of  
4 the root node are the top level XML tags “menu” and “toolbars”. The Filter  
5 attachment point uses the Explode attachment point as its data source and filters  
6 the exposed objects looking only for the nodes that are “menus”. The outer  
7 Explode attachment point uses the Filter attachment point as its data source and  
8 exposes all of the children of the filtered menu nodes to provide a list containing  
9 the two menus that are added by the extension. Since this particular XML file  
10 contained menus that were identified by the attachment points associated with  
11 menu attachment manager, that attachment manager is then notified that two  
12 menus have been added by an extension.

13 This process is diagrammatically illustrated in Fig. 19 which shows  
14 attachment points 1900, 1902, 1904, and 1906. Each attachment point exposes a  
15 list of XML nodes. URL attachment point 1900 takes an input (a URL to an XML  
16 file—e.g. *retail.edf*) and exposes a list of XML nodes. This list contains only the  
17 root node “<edf:extension>”. Explode attachment point 1902 takes as an input  
18 attachment point 1900 and exposes a list of XML nodes which are children of  
19 source XML nodes. In this example, the list of XML nodes exposed by  
20 attachment point 1902 are the “<menus>” nodes and the “<toolbars>” nodes. The  
21 filter attachment point 1904 takes attachment point 1902 as an input and filters on  
22 “menus.” It then exposes an XML list having only “<menus>” nodes in it. The  
23 explode attachment point 1906 takes attachment point 1904 as an input and  
24 exposes a list with the XML nodes that are contained in the “<menus>” nodes—  
25 here both of the “<menu>” nodes.

1       Consider additionally that the toolbar attachment manager would request a  
2 predicate chain of attachment points which would also use URL attachment point,  
3 an Explode attachment point and a filter attachment point 1904 that filters on  
4 “toolbars”. Thus, the corresponding explode attachment point 1906 would expose  
5 an XML list containing only the “<toolbar>” node. But, the attachment point  
6 manager would detect the commonality of the URL attachment point and the inner  
7 Explode attachment point, so it would reuse the same attachment points it created  
8 for the menu attachment manager. The Filter attachment points used by the  
9 toolbar attachment manager and the menu attachment manager would use the  
10 same Explode attachment point as their data sources but would expose different  
11 collections of nodes, because they were filtering based on different criteria.

12       Consider Fig. 20 which incorporates an EDFHub attachment point 2000.  
13 This attachment point receives all of the EDFs and, as discussed above, combines  
14 them into a single XML list. The EDFHub then exposes the root nodes of all of  
15 the EDFs. The explode attachment point 2002 then exposes an XML list that  
16 contains all of the top level nodes for all of the EDFs. As an example, there may  
17 be multiple EDFs that each contain top level menu nodes, toolbar nodes,  
18 accelerator nodes and the like. Explode attachment point 2002 exposes an XML  
19 list that contains all of these top level nodes for all of the EDFs. Filter attachment  
20 point 2004 can then filter the XML list exposed by the explode attachment point  
21 2002 in accordance with any suitable parameters (i.e. filter on menu nodes, tool  
22 bar nodes, accelerator nodes and the like). The final explode attachment point  
23 2006 then exposes a list of the individual children nodes of the list exposed by the  
24 filter attachment point 2004. This list describes all of the specific features (of the  
25 particular type that were filtered) that have been added by all of the EDFs.



The table below lists a number of different attachment points that can be utilized in accordance with this described embodiment but many more can easily be created.

Attachment Point	Purpose
URL	Loads the URL and exposes the top level XML node as a member of the collection
Context	For every member, it gets the "expression" attribute and binds to it. If the expression evaluates to true, then the member is exposed.
EDF	Same as the URL AP, but also exposes a fabricated member with data to create an APP based on the URL and URN (which exists in the XML DOM).
Merge	Takes zero or more Attachment Points (of any type) and merges them together. The order and continuity of the original collections will be maintained.
Filter	Monitors a single Attachment Point and only exposes those nodes that match the specified name. The order of the original collection will be maintained.
Duplicate	Monitors a single Attachment Point and filters out any duplicates. A duplicate is defined to be having the same URN attribute. If no URN attribute is present then the node is exposed. Order of the original collection will be maintained.
Explode	Monitors a single Attachment Point and for every member exposes the children of that member as its members. The order of the original collection will be maintained as well as the order of the children within the nodes.
Link	Monitors a single Attachment Point and for every member looks for a URL attribute and creates a URL AP and

	merges it into itself. If the optional include Content is set to true, it will merge the original collection in as well.
Order	Monitors a single Attachment Point. For every member, it gets three attributes: id, before and after. Based on this information, it reorders the members as specified. If no ordering information is supplied, the order of the original collection will be maintained.
EDFHub	This Attachment Point is the central merge point that represents all the EDF points.

### **Conclusion**

The embodiments described above provide a platform solution that provides for customization and extensibility through a consistent and logical extensibility mechanism and object model that can be easily understood by third party developers. Internet-based downloads can be accomplished without a great deal of user intervention and without manipulating any user persisted settings. Extensions can be provided to a software platform or application program dynamically based upon the user's computing context.

Although the invention has been described in language specific to structural features and/or methodological steps, it is to be understood that the invention defined in the appended claims is not necessarily limited to the specific features or steps described. Rather, the specific features and steps are disclosed as preferred forms of implementing the claimed invention.